Hybrid-SIMD: a Modular and Reconfigurable approach to Beyond von Neumann Computing

Hybrid-SIMD: a Modular and Reconfigurable approach to Beyond von Neumann Computing

Andrea Coluccio, Umberto Casale, Angela Guastamacchia
Giovanna Turvani, Marco Vacca, Massimo Ruo Roch, Maurizio Zamboni and Mariagrazia Graziano

Abstract—The increasing complexity of real-life applications demands constant improvements of microprocessor systems. One of the most frequently adopted microprocessor design scheme is the von Neumann architecture. Central Processing Unit (CPU) performs computations and communicates with memory in a constant exchange of information. This unceasing motion of data between these two components became a significant performance bottleneck. A lot of power, energy, and computational time are wasted in this communication. With Beyond von Neumann Computing (BvNC) paradigms, calculations are performed inside or very close to a memory array. BvNC approaches are proposed in the literature, mainly based on modifications of existing memories, enabling simple computations. Others exploit emerging technologies to both store and compute data, using analog operations.
In this work we follow a different approach, where computational units are placed close to memory cells, improving versatility and performance. We propose a Hybrid-SIMD architecture made of memory and computing elements in an interleaved structure. Hybrid-SIMD can be used both as a low density memory and as SIMD accelerator. We insert our design in a classical von Neumann system based on a RISC-V processor, and we estimate its impact, demonstrating its capability to improve speed reducing at the same time energy consumption.

Index Terms—Memory Wall, Beyond von Neumann Computing, In-Memory Computing, Near-Memory Computing, SIMD

1 Introduction

The increasing complexity of real-life applications requires very high computational capabilities and more complex devices, implying stringent constraints on microprocessors design. CMOS-based microprocessors have made significant performance improvements, becoming faster and more efficient over the years. Most recent CPUs are designed to exploit Thread-Level Parallelism (TLP) and to achieve very high frequencies while keeping at the same time a low operating power. Most CPU architectures follow the Von Neumann structure: it is made of a CPU and memory that are well separated from each other and connected through a bus. CPU is in charge of performing the computations, while memory provides the data in a constant exchange of information. However, this unceasing motion constitutes the so-called von Neumann bottleneck or memory-wall problem: a lot of energy, power, and computational time are wasted only to move data from one another [1]. In the literature, several solutions are proposed to overcome the memory-wall problem. Out-Of-Order (OOO) scheduling allows the execution of concurrent instructions in the pipeline during a memory operation, covering the communication latency by performing other algorithms computations. Other techniques consist of high-speed cache memories and prefetching, but also in these cases, the von Neumann bottleneck cannot be solved entirely.

Beyond von Neumann Computing (BvNC) architectures represents a promising solution to this problem. A completely new way of designing CPU-centric architectures is currently studied by researchers. Part of the computation is moved inside or very close to a memory array, reducing the data traffic. If the computation is performed inside the memory array, the methodology is called In-Memory Computing, otherwise Near-Memory Computing. The In-Memory Computing approaches are generally based on the use of different memory technologies. In standard CMOS, the most frequently adopted solutions are based on SRAM [2], [3], [4] and DRAM [5], [6] arrays. They consist mainly on enabling multiple wordlines (WLs) at the same time and sensing the voltage potentials on the bitlines (BLs). BLs voltages will depend on the contents of the considered cells, so by choosing an appropriate threshold voltage, the sense amplifier output will coincide with a logic function.

Similarly, also the BL current can be used to perform a current-based computation. The BL current is proportional to the conductance on the path, so it depends on the cell transistors state. This methodology is exploited in [7], in which authors obtained a multibit dot-product 8T SRAM engine. Moreover, SRAM-based computing is an advantageous technique exploited in [8] to empower cache memories, allowing logical computations within the array and arithmetic operations with a small Near-Memory computational unit. CAM memories can also be used to perform calculations since they execute the XOR operation between BL and the cell content. The result will be written on the Match Line (ML), which is initially precharged to Vdd: if there is no match, the XOR result is 0, and the ML will be discharged to 0. This working principle is adopted in associative processing [9], or in XNOR-Binary Neural Networks [10], where weights and inputs can assume only two values (± 1). In this way, the multiply-and-accumulate (MAC) operation is transformed into an XNOR-

• *Authors are with the Department of Electronics and Telecommunication, Politecnico di Torino, Italy*
E-mail: giovanna.turvani@polito.it

Manuscript received

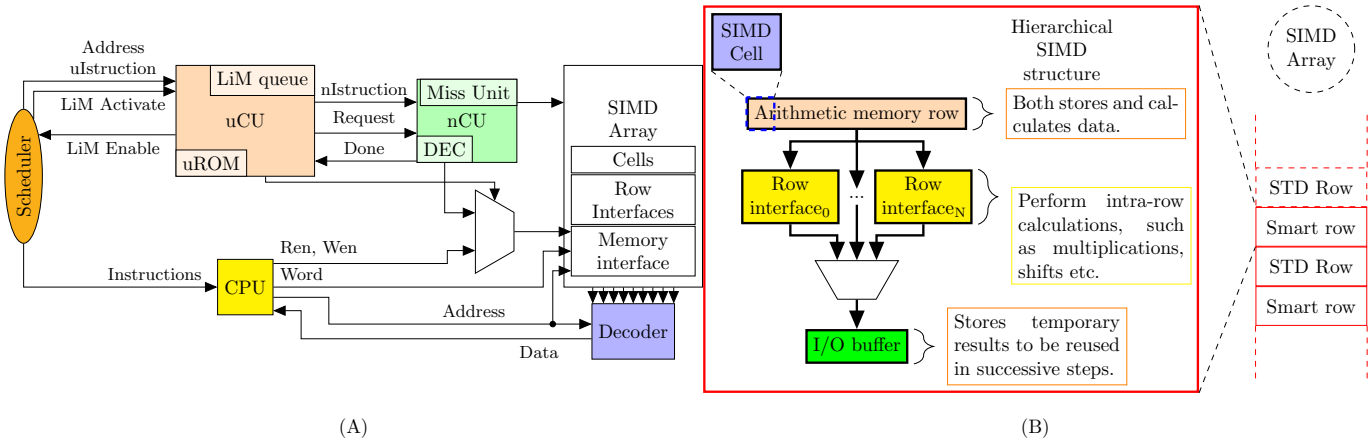

Fig. 1. (A) Hybrid-SIMD high level system overview. (B) SIMD Array structural overview.

Population counting sequence [11]. Another important In-Memory Computing approach consists on the use of emerging resistive technologies, very popular in the In-Memory computing field, such that they become a promising alternative to CMOS arrays. Technologies like Magnetic Tunnel Junction (MTJ) [12], [13], [14],Memristors [15], [16], [17], [18] and Phase Change Memories (PCMs) [19], [20], [21] are often used instead of CMOS transistors. They are capable of both storing data and performing computations within the array, maintaining low operating power and high efficiency at the same time. Such devices store data through their intrinsic resistance, which can assume a high or low value depending on the applied voltage or current: high resistance (R_H) is usually associated to logic '0', while low resistance (R_L) to logic '1'. Usually, the computation is performed by exploiting Kirchhoff's current laws, similarly to current based computation in standard technologies. The categories explained so far can execute simple logical or arithmetical computations in a highly parallelized fashion.

More complex operations must be delegated to an external processing unit: this happens in Near-Memory computing architectures, such as Hybrid Memory Cubes (HMCs), where a processing unit is put very close to the memory. Usually, HMCs are based on DRAM memories, and the intrinsic cell structure is kept unaltered. DRAM arrays are organized in vertically stacked layers connected utilizing Through Silicon Vias (TSVs) to a logical one [22]. Using TSVs shortens the data path and achieves a very high bandwidth. The concept of Near-Memory and In-Memory computing can also be extended to Single-Instruction-Multiple-Data (SIMD) acceleratorsas in [9], [23], [24], where SIMD processing units are put very close or inside a memory array. In [9], authors presented an associative processor that replaces the last level of cache, enabling both SIMD computation and storage capabilities at the same time, and [23] realizes a SIMD unit closely connected to a cache, improving parallel computations and enabling re-usability of the design.

In this paper, we concentrate on the last BvNC category, and we propose a Hybrid-SIMD accelerator to reduce the data traffic between CPU-Memory. The high versatility and computational capabilities of a SIMD accelerator and the closeness to the memory leave room for huge improvements in von Neumann architectures, convincing us to investigate more on this architectural solution. Consequently, we propose a hybrid structure, made of both memory and computational elements, organized in stacks and capable of acting both as a standard low density memory and a SIMD accelerator. The modular approach that we propose, can be used to create custom architectures for one specific algorithm, or it can be used to create a reconfigurable architecture capable of accelerating several algorithms. We focus our work on the second option. Our goal is to evaluate the impact of our Hybrid-SIMD coprocessor on a classical von Neumann scheme, testing it with several benchmarks. Performance results are then compared with a standard RISC-V with standard memories, demonstrating the Hybrid-SIMD capability to reduce the von Neumann bottleneck. The rest of the paper is organized as follows: section 2 describes the architecture structure in detail. Section 3 illustrates the chosen benchmarks and how to map them on our Hybrid-SIMD, section 4 reports the performance results and section 5 evaluates the differences between von Neumann/BvNC Hybrid-SIMD systems. Lastly, section 6 concludes the paper.

2 Architecture

In this section, the architecture is explained in detail. The main parts of the Hybrid-SIMD are depicted in Figure 1 (A).

- SIMD Array: stores and computes data. It can be used both as a standard memory and a SIMD computational unit;
- Decoder: extracts data from the SIMD Array;
- Controlling part: made of a micro-Control Unit (uCU) and a nano-Control Unit (nCU);
- CPU: computes the instructions that are not delivered to the SIMD Array. It can send data to the SIMD Array (indicated as Word in the scheme) using the memory interface, which stores the incoming Word to the memory address pointed by the Address signal. Data from the SIMD Array is extracted employing the decoder.
- Scheduler: provides all the signals required by the architecture to work properly. This component has been implemented as a part of the UVM testbench.

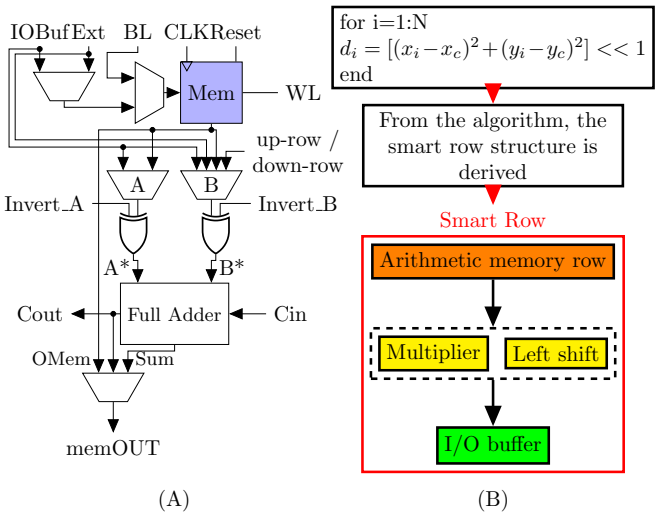

Fig. 2. (A) Hybrid-SIMD cell structure. (B) Algorithm driven approach to choose the row intefaces inside a smart row.

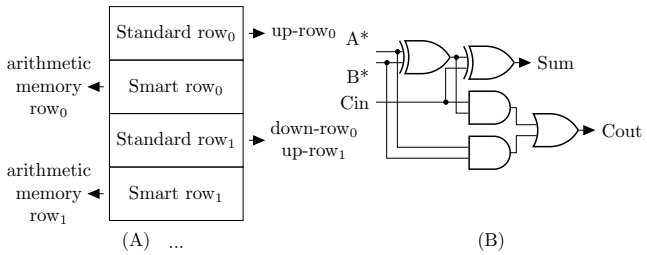

Fig. 3. (A) SIMD Array structure: smart and standard rows interleaved scheme. (B) Schematic of the full adder.

Each element of the Hybrid-SIMD architecture will be discussed in detail in the following parts.

2.1 SIMD Array

The SIMD Array in Figure 1 (A) is the core of the Hybrid-SIMD architecture. It performs calculations and stores data, reducing data traffic between the CPU and the memory itself. The elements composing the SIMD Array are cells, row interfaces, and memory interface: a high-level scheme of the cells-row interfaces organization of the SIMD Array is depicted in Figure 1 (B). One can notice that the structure is organized hierarchically: cells compose memory rows that, combined with row interfaces and I/O buffers, build a smart row. The SIMD Array is obtained by stacking multiple smart row and standard rows together. A standard row memorizes data without manipulation, so it can be considered a classical memory. Inside a smart row, calculation and storage tasks are performed, and the computation capability of a smart row depends on the row interface functionality. Since it is a modular design, a smart row can be easily configured, meaning that each smart row building block has the same interface. Depending on the application, the row interfaces are chosen accordingly. As shown in Figure 1 (B), the SIMD Array is composed of smart rows and standard ones in an interleaved fashion. Standard rows have a fundamental role in the SIMD computation, holding local data used by the smart rows.

TABLE 1
Available operations performed by the full adder in the SIMD cell. For the sum, the full adders of each cell are connected to form a ripple carry adder.

Operations	Cin value	Output pin
A* OR B*	1	Cout
A* AND B*	0	Cout
$A^* \oplus B^*$	0	Sum
$\overline{A^* \oplus B^*}$	1	Sum
A* + B*	Cout(i-1)	Sum/Cout

There are two standard rows for each smart row and, as shown in Figure 3 (A), they are called up-rows and down-rows, respectively. These last ones are shared between two consecutive smart rows, so the first smart row down-row becomes the up-row of the second smart row. This structure allows moving data from one smart row to another and it is called up-row/down-row mechanism.

2.1.1 Cells

Cells are the finer-grained building blocks of the array. A cell contains the storage element and some logic gates to execute calculations. The structure is shown in Figure 2 (A). The storage element (denoted as Mem in the figure) is a flip-flop, and it takes three possible values in input: bitline (BL), Input/Output Buffer (IOBuf), and external (Ext). BL comes directly from the CPU, which streams data to be memorized inside the SIMD Array, while IOBuf and Ext are reserved to the data stored in the SIMD Array itself. IOBuf is connected to the Input/Output buffer (depicted in Figure 1 (B)) of the same smart row to reuse temporary data of a computation. Ext is connected to the memory interface, allowing to fetch data from the SIMD Array stored in any memory location. Inside each cell, there are two XOR gates, a full adder and some multiplexers. The XOR gates invert the A and B values based on the input signals Invert_A and Invert_B; the full adder computes different kinds of logical functions and the arithmetical sum of its inputs A* and B*, based on the value of the input Cin. Considering the SIMD cell structure in Figure 2 (A) and the full adder scheme in Figure 3 (B), the available operations are reported in Table 1. Output pin specifies which output of the full adder between Cout and Sum describes the target operation. Other logical and arithmetical functions (NAND, NOR, Subtraction, etc.) can be easily obtained utilizing Invert_A and Invert_B. If, for instance, a NOR operation is required between A and B, it is sufficient to exploit A* AND B* with A* = $\overline{A}$ and B* = $\overline{B}$. Similarly, A-B is obtained with A*+B* where A* = A and B* = $\overline{B}$ and Cin(0) = 1. The arithmetic memory row, shown in Figure 1 (B), is obtained by instantiating Nbit cells together, where Nbit is the data parallelism specified at design phase.

2.1.2 Row interface

The row interface performs intra-row calculations. The computations that cannot be executed by an arithmetic memory row (different operations from arithmetic sum/subtraction and logical functions) are delegated to row interfaces that must follow common design guidelines to meet reconfigurability and modularity requirements of the Hybrid-SIMD. The row interfaces are chosen according to the required operations in the algorithm, as shown in the example in

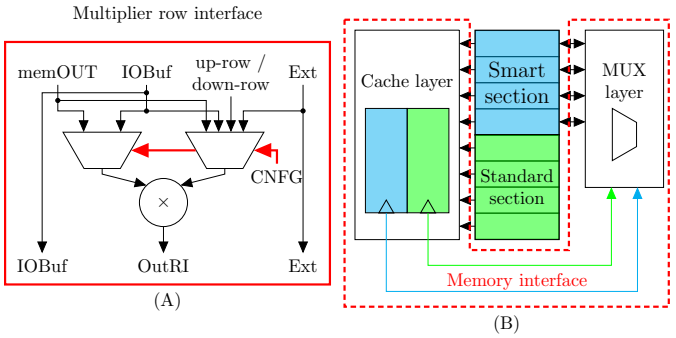

Fig. 4. (A) High-level scheme of a multiplier row interface. (B) High-level scheme of the memory interface.

Figure 2 (B): the algorithm involves sums/subtractions that can be handled by the arithmetic memory row. Power-2 and shift computations necessarily require a multiplier and a shift left row interfaces. One can notice that row interfaces, I/O buffer and arithmetic memory row must be designed in order to be interconnected together: from these considerations, common interface becomes a design constraint. In Figure 4 (A), a multiplier row interface is presented. There are four data inputs denoted as arithmetic memory row output (memOUT), I/O buffer (IOBuf), up-row/down-row and external input (Ext).

The memOUT and IOBuf inputs come from the arithmetic memory row and the smart row input/output buffer (shown in Figure 1 (B)), respectively. The up-row/down-row input comes from the up-row or the down-row, depending on the operation and finally, Ext is the data provided by the memory interface in a specific SIMD Array address. The multiplexers in Figure 4 (A) select the inputs combinations according to Table 2, which specifies all the possible combinations of inputs that the Hybrid-SIMD can compute. For instance, a multiplication between the up-row and the output buffer source operands (UpR in Table 2) is accomplished by the multiplier row interface by selecting memOUT and IOBuf as inputs. Once row interfaces have been designed, they must be connected following the pattern reported in Figure 1 (B). The connections are straight-forward because the interfaces are the same: this is a solid point of our Hybrid-SIMD architecture because, at the design phase, the user can insert or remove the row interfaces with high flexibility.

2.1.3 Memory interface

To reduce the complexity of the design, the SIMD Array is split in half, as shown in Figure 4 (B). The first half is the smart section containing smart rows and standard rows interleaved following the pattern in Figure 1 (B). The second one is the standard section, which is only made of standard rows. The connections between smart and standard sections are handled by the memory interface, composed of cache and MUX layers. The possibility to access the entire space memory address is fundamental: in this way, data can be fetched from any possible SIMD Array location, and associated to the Ext operand. The cache layer contains two registers that hold data of smart and standard sections, as shown in Figure 4 (B). The usage of these two registers reduces the design critical path by separating the MUX layer from cache one. The MUX layer simply connects the cache registers to the smart section, using several multiplexers.

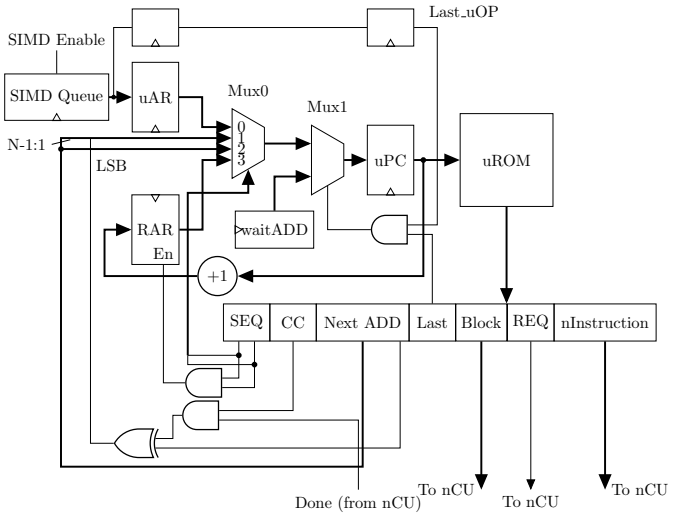

Fig. 5. Microprogrammed uCU scheme

2.2 Control logic

The control logic includes a micro-Control Unit (uCU) and a nano-Control Unit (nCU). From a hierarchical point of view, the uCU takes high-level assembler-like instructions of algorithmic procedural step (called nano-Instructions or nInstructions), which are then translated into low-level signals by the nCU to handle the SIMD Array. This organization eases the user's work in implementing and debugging the algorithms on Hybrid-SIMD, reducing the design complexity to simply choosing architectural metrics (data parallelism, SIMD Array dimensions, etc.) and writing high-level nInstructions. Regarding the uCU, a microprogrammed machine is employed to improve flexibility and reusability. The structure is depicted in Figure 5: basically, uCU is composed of a microprogram counter (uPC) that is in charge of fetching a uROM address; a uROM, and a control logic part that selects the next input of the uPC.

At the beginning of the algorithm, the instruction address is fetched from the SIMD Queue and stored inside the micro Address Register (uAR). This content will be sent to the uPC, which selects the uROM line with the first address. When the algorithm starts, the uCU extracts uROM data in each clock cycle. Each uROM row contains the following fields:

- SEQ: instruction addressing mode, sequential, or branch. Branches are performed with the Return Address Register (RAR) that takes the uPC value incremented by 1, allowing to return from a subroutine;
- Condition Code (CC): used to perform branches. Looking at the Mux0 1st input in Figure 5, only the LSB is affected by CC and Done. When both are set to 1, a branch is performed to the previous or next uROM instruction;
- Next ADD: next address of the uROM;
- Last: last uROM instruction flag. When 1, the uPC fetches the wait address from the waitADD register. Normally, the next address is extracted from the

"Next ADD" of the uROM, but when the last uROM address is reached, a branch will be taken to the wait address, which signals the computation end.
- REQ: request flag sent to the nCU to start the conversion from the high-level assembler-like nInstrution to SIMD Array low-level controlling signals;
- Block: which block is activated in a specific algorithm step;
- nInstruction: a high-level description of the computation that must be performed in a specific algorithm step. The nCU translates this field.

2.2.1 Block

Since the Hybrid-SIMD is a SIMD architecture, each smart row should execute the same nInstruction. To reduce the dynamic power consumption of the design, the SIMD Array has been divided into blocks that can be switched off when not involved in the computation. Each block contains N smart rows, where N is defined at the design phase by the user. N could be any value between 1 and NSmartRow, where NSmartRow is the total number of smart rows present in the SIMD Array. If N is equal to 1, it implies the finest design granularity level since the computation can be potentially executed by a single smart row. On the other hand, if N is equal to NSmartRow, all smart rows execute the computation.

2.2.2 nInstruction

The most important uROM data is nInstruction, which is composed of the following fields:

- Opcode: operating code, indicating which row interface is activated at a specific step;
- Source: the source operands of the computation;
- Store location: where to store the results;
- Source address: needed in case of computations that require a specific data stored inside another SIMD Array address;
- Function: specifies the required function, logical or arithmetical. This field is NULL when no arithmetic or logical operations are performed in the considered algorithmic step.

These values have a correspondent binary code described in a VHDL package. To better understand the nInstruction philosophy, the smart row shown in Figure 6 is considered as an example.

The operation $OBUF = (A + B)$ is performed by enabling the arithmetic memory row and the I/O buffer block to store the temporary result. The opcode indicates which block performs the computation, so it becomes:

```
Opcode: Arithmetic_memory_row
```

assuming that the operands A and B come from the arithmetic memory row and a specific memory address location of the SIMD Array respectively, the source operand is written as:

```
Source: Ext
```

where Ext stands for "external". Other possible source operands are reported in Table 2. In Table 2, the term "external data" refers to that data that comes from any possible memory address, so it is external from the smart row environment. This operand is provided by the memory interface, as discussed in subsubsection 2.1.3.

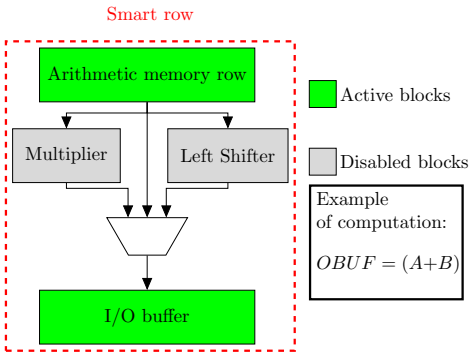

Fig. 6. Example of a computation involving a sum. In this example, the smart row contains an arithmetic memory row, a multiplier, a left shifter, and an I/O Buffer. Blocks can be enabled/disabled according to the requested operation.

TABLE 2
Available source operands for nInstruction. "Row" refers to the arithmetic memory row.

Source operands	Meaning
Row	Operation is performed between row — row (e.g. power-2 computation)
UpR	Row — up-row operation
DwnR	Row — down-row operation
Bo_UpR	Output buffer — up-row operation
Bo_DwnR	Output buffer — down-row operation
Bo_Ext	Output buffer — External data operation
Bi_UpR	Input buffer — up-row operation
Bi_DwnR	Input buffer — down-row operation
Bi_Ext	Input buffer — External data operation
Ext	Row — External data operation
OBuff	Row — Output buffer operation
IBuff	Row — Input buffer operation
Bo_Bo	Output buffer — Output buffer operation (similar to Row)
Bi_Bi	Input buffer — Input buffer operation

The store location is the output buffer in the I/O buffer block, so:

```
Store location: OutputBuffer
```

the source address specifies the address value "AddressB" of the operand B, so it assumes a numeric value.

```
Source address: AddressB
```

This field is only used in Bo_Ext, Bi_Ext, and Ext source operands because it specifies the memory address from which the external data value is taken. Notice that only one source address is required because each case consider only one external data at a time. For Bo_Ext, Bi_Ext and Ext, the other data come from the output buffer, the input buffer or the arithmetic memory row (Ext) of the same smart row, respectively. Finally, the function field specifies the operation performed by the arithmetic memory row, which in this case is a sum.

```
Function: SUM
```

In Hybrid-SIMD design, a set of special-purpose nInstructions is defined.

(1) Multiplier&Ext&StoreBuff&ADDRESS_2PI&nullFunc;
(2) Multiplier&Bo_Ext&StoreBuff&ADDRESS_K&nullFunc;
(3) ShiftRight&Bo_Ext&StoreBuff&ADDRESS_N&nullFunc;

(4) LUT&OBuff&StoreBuff&nullAdd&nullFunc;(1st half)
(5) LUT&OBuff&StoreBuff&nullAdd&nullFunc;(2nd half)
(6) Multiplier&Bo_UpR&StoreBuff&nullAdd&nullFunc
(7) Store&ROW&null&nullAdd&nullFunc;
(8) Load&UpR&null&nullAdd&nullFunc;

(9) Store&DwnR&null&nullAdd&nullFunc;
(10) Arith_memory_row&UpR&StoreBuff&nullAdd&SUM;

Fig. 7. nInstructions for DFT algorithm

- StoreI: takes data from the input buffer (located in the I/O buffer row interface) and stores it in the down-row or up-row, depending on the source operand value;
- Store: moves data from the output buffer (located in the I/O buffer row interface) to the down-row or up-row, depending on the source operand value;
- Load: moves data inside the input buffer, depending on the source operand. The possible source operands are a subset of the ones specified in Table 2: UpR, DwnR, and Ext.

The nCU, in addition to nInstructions translation, checks for misses of the memory interface in the cache layer. If data held by the registers, shown in Figure 4 (B), do not correspond to the desired ones, contents are discarded and written again in the registers. This mechanism requires two clock cycles of latency for each fetched nInstruction.

3 Algorithm Mapping

To test SIMD Array capabilities, different applications are chosen as benchmarks. This section discusses how Hybrid-SIMD executes nInstructions, and the design flow adopted: starting from the application, the smart row structure is derived and the algorithm is manually described in terms of assembler-like nInstructions.

3.1 Discrete Fourier Transform

The first proposed benchmark is the Discrete Fourier Transform (DFT). The algorithm consists of performing the following computation:

$$X_k = \sum_{i=0}^{N-1} x_i \times \left[cos\left(\frac{2\pi ik}{N}\right) - jsin\left(\frac{2\pi ik}{N}\right)\right] \quad (1)$$

where N is the total number of samples and k is the considered element. From Equation 1 and following the scheme in Figure 2 (B), the structure of the smart row is immediately derived. It contains an arithmetic memory row (to compute the sum), a multiplier and a look-up table (LUT) to store cosine-sine values. Regarding the division, since the divider is a very complex computational element, inserting it for each smart row would degrade the performance. For this reason, we choose N such that $N = 2^p$, so the division result

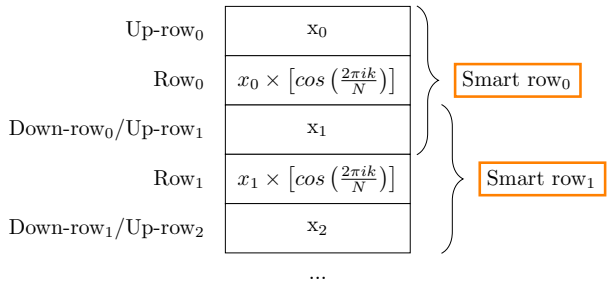

Fig. 8. SIMD Array content of the DFT algorithm until point (8)

can be obtained by simply right shifting the operand by p positions. This represents a limiting operative constraint to the Hybrid-SIMD because more complex division computations must be delegated to the CPU. Referring to Figure 7, the algorithm starts computing all the $\left(\frac{2\pi ik}{N}\right)$ terms and by saving the results in the output buffers. Two multiplications and a right shift are performed. The first multiplication, labeled as (1) in Figure 7, computes $2\pi \times i$. These operands are saved in ADDRESS_2PI and in the arithmetic memory rows, respectively. The second multiplication (2) involves the previous results (stored inside the output buffers) and the term K (stored in ADDRESS_K of the SIMD Array). Also in this case, the computation outcomes are saved in the output buffers, which are reused by the arithmetic right shifter block to compute the divisions with (3).

To obtain the real and imaginary parts, the SIMD Array smart section is split in half. The first part is reserved to the real terms of the DFT, so it computes the cosine of $\left(\frac{2\pi ik}{N}\right)$ with (4), while the second one the imaginary part with the sine of the same value with (5). The splitting operation can be handled by exploiting the Block signal of the uCU. From now on, we consider only the first half content of the SIMD Array since the operations performed on real and imaginary parts are exactly the same. At this point, the output buffers of the first half contain the cosine terms, while the sine terms are in the second half. In (6) the samples x_i are multiplied in the real and imaginary case. Since x_i are stored in the up-rows, the corresponding nInstruction has Bo_UpR as source operand field. For brevity, in Table 3 the term $x_i \times cos\left(\frac{2\pi \times i \times k}{N}\right)$ is replaced with $A(i,k)$ in the first half. The output buffers contents are now moved into the arithmetic memory rows with (7). Until now, Hybrid-SIMD performed the operations required to calculate each contribution of the sum in Equation 1. The SIMD Array content is shown in Figure 8: each contribution is stored inside each arithmetic memory row and must be added together in order to find the final X_k result. To do this, up-row/down-row mechanism is exploited: the down-row of the i-th smart row becomes the up-row of the i+1-th smart row, so two adjacent smart rows can exchange data. However, the sum performed with the up-row/down-row mechanism destroys the content of the up-rows x_i that is required for a successive iteration. To avoid data loss, a load is performed at point (8), where the contents of the up-rows are stored inside the input buffers in the I/O buffer row interfaces. Similarly to point (7), at point (9) a store operation is performed such that the contents of the output buffers are moved inside the down-rows: in this way the down-rows have the same data. The first sum iteration

is computed at point (10). Up-row/down-row mechanism performs an addition between the rows and the up-rows. Consequently, the contents of the output buffers become:

$$\text{Output buffer}_0 = x_0 + x_0 \times \left[cos\left(\frac{2\pi ik}{N}\right)\right]$$

$$\text{Output buffer}_1 = x_0 \times \left[cos\left(\frac{2\pi ik}{N}\right)\right] + x_1 \times \left[cos\left(\frac{2\pi ik}{N}\right)\right]$$

...

The output buffers contain the sum between the i-th and (i-1)-th element. According to Equation 1, by simply iterating N-1 times points (9) and (10), the final sum of Equation 1 can be achieved. Table 3 summarizes the algorithm implementation by reporting the smart row contents for each iterative step. The algorithm requires $2 \times N + 7$ steps to be completed.

3.2 K-Means

The second proposed benchmark is K-means [25]. K-means aims to find a predefined number Nc of clusters. The distances between each dataset point and Nc centroids are computed, choosing the point with the minimum distance as belonging to the cluster. In this paper, a single iteration of the clusterID identification is considered, meaning that the new centroid calculations are not performed. The high-level description of the algorithm is the following:

```
d(:,0) = MAX_N;
for j in Nc centroids {Xc(j), Yc(j)}:
    for i in 0 to #nodes:
        d(i,j+1) = (X(i) - Xc(j))^2 +
                 + (Y(i) - Yc(j))^2;
        if d(i,j+1) < d(i,j):
            clusterID(X(i),Y(i)) = j;
        else:
            d(i,j+1) = d(i,j);
        end
    end
end
```

From the algorithm description, the smart row structure is derived, which contains an arithmetic memory row, a multiplier (for power-2 computation), and a conditional block. This last one is a comparator that returns the lowest value between its two inputs, so it computes *output = (A<B) ? A:B*. This operation is implemented as a subtraction between the two input terms A and B. If the result is negative, the condition is true and a multiplexer driven by the MSB of the result forwards A in output, otherwise B. One can notice that, for each point in the dataset, a set of distances from Nc centroids should be computed, and if each point is stored for each smart row, there is not a sufficient number of storage elements to keep all the required local data. To solve this problem, a new row interface named TemporaryStorage is used. It is a register that acts as a storage element, keeping all the requested data inside each smart row: if there are Nc centroids then, to keep Nc distances, Nc TemporaryStorages are required.

Initially, the up-rows and arithmetic memory rows contain X(i) and Y(i) respectively, while centroids nodes are stored in the standard section of the SIMD Array. The algorithm steps are the following:

1) Up-rows contents are copied inside input buffers to avoid data loss in up-row/down-row mechanism.
2) Subtractions between X(i) and Xc(j) are performed, saving the results in the output buffers.
3) The multiplier blocks are used to perform power-2 computations of the output buffers contents. The results are stored in the output buffers again.
4) The output buffers contents are moved in the down-rows.
5) Similarly to points 2-3, $[Y(i) - Yc(j)]^2$ are computed and stored inside the output buffers.
6) The final sums between $[X(i) - Xc(j)]^2$ and $[Y(i) - Yc(j)]^2$ are performed and the results are saved in the output buffers.
7) An OR operation is performed between the contents of the output buffers (so the final sums) and a MASK_j (written at the address ADDRESS_MASK_j of the SIMD Array). The MASK_j value depends on the considered centroid, and it is used to define the clusterID explicitly. Its definition is `MASK_j = j << Nbit`, where Nbit is the data parallelism and j is the clustedID index. MASK_j is all zeros, except for the MSBs. By ORing the just computed final sums with the MASK_j, each sum is exclusively associated with a cluster. This solution requires $log_2(Nc)$ additional bits to the original Nbit value.
8) The output buffers data represent the distances. These values are copied inside the TemporaryStorages(j), which will be reused for comparation later.

Points 2-8 are repeated for Nc times until all the distances have been computed. TemporaryStorages(0-Nc) will contain all the distances between the Nc centroids plus the $log_2(Nc)$ bits dedicated to the clusterIDs. At this point, the remaining operations are:

9) The contents of TemporaryStorages(j-1) are moved to the output buffers.
10) Output buffers contents are stored inside the down-rows.
11) TemporaryStorages(j) are copied inside output buffers.
12) The comparators compare output buffers and down-rows contents and save the result in the output buffers.
13) Output buffers contents are stored in the down-rows, which contain the smallest resulting distance.

With steps 9-11, Hybrid-SIMD moves the first two distances from the TemporaryStorages into the output buffers and down-rows, respectively, to perform the comparation at point 12. Points 11-13 are repeated for Nc-1 times to scan all the distances stored in the TemporaryStorages. In the final part of the algorithm, the down-rows will contain the minimum distance in the Nbit LSBs and the clusterID in the $log_2(Nc)$ MSBs. The algorithm requires $11 \times N_c$ steps to be completed.

3.3 Matrix Vector Multiplication

The computation performed in the Matrix Vector Multiplication (MVM) is the following one:

$$\overline{Z} = \overline{\overline{X}} \times \overline{Y} \text{ where } \overline{\overline{X}} \in \mathbb{R}^{N \times M}, \overline{Y} \in \mathbb{R}^{M \times 1}, \overline{Z} \in \mathbb{R}^{N \times 1} \quad (2)$$

TABLE 3
Content of the smart rows for each iterative step with $A(i,k) = x_i \times cos\left(\frac{2\pi \times i \times k}{N}\right)$ (first half)

Smart row	Iterative steps: Init	1	2	3	4	5	6	7	8	9	10
Up-row	x_i	x_i	x_i	x_i	x_i	x_i	x_i	x_i	x_i	$A(i-1,k)$	$A(i-1,k)$
Row	i	i	i	i	i	i	i	$A(i,k)$	$A(i,k)$	$A(i,k)$	$A(i,k)$
Down-row	x_{i+1}	x_{i+1}	x_{i+1}	x_{i+1}	x_{i+1}	x_{i+1}	x_{i+1}	x_{i+1}	x_{i+1}	$A(i,k)$	$A(i,k)$
Input buffer	-	-	-	-	-	-	-	-	x_i	x_i	x_i
Output buffer	-	$2\pi \times i$	$2\pi \times i \times k$	$\frac{2\pi \times i \times k}{N}$	$cos\left(\frac{2\pi \times i \times k}{N}\right)$	$sin\left(\frac{2\pi \times i \times k}{N}\right)$	$A(i,k)$	$A(i,k)$	$A(i,k)$	$A(i,k)$	$A(i,k) + A(i-1,k)$
SIMD Array section	All	All	All	All	First half	Second Half	First half	First half	First half	First half	First half

From Equation 2, the smart rows contains only an arithmetic memory row, a multiplier, and I/O buffer row interfaces. Initially, $\overline{Y}$ and $\overline{\overline{X}}$ are stored inside the up-rows and arithmetic memory rows, respectively. The algorithm starts saving all $\overline{Y}$ values inside the input buffers to free the up-rows/down-rows (1 nInstruction required). Then, the punctual multiplication between $\overline{Y}$ and $\overline{\overline{X}}$ is performed, saved in the output buffers, and moved inside the rows/down-rows (2 nInstructions required). Finally, by exploiting the up-row/down-row mechanism, the multiplications are propagated along the SIMD Array to perform the partial sums ($2 \times (M-1) + 1$ nInstructions required). The algorithm requires $4 + 2(M-1)$ steps to be completed.

3.4 K-NN

K-Nearest Neighbor is one of the simplest classification and regression algorithm that aims to find k closest points to a reference node. The main part of the algorithm consists of computing the distance as $D_i = |x_s - x_i| + |y_s - y_i|$, where (x_i, y_i) is the considered point, while (x_s, y_s) is the reference node. This algorithm can be easily performed with Hybrid-SIMD by inserting an absolute value row interface (ABS). As before, x_i and y_i are stored inside the up-rows and arithmetic memory rows, respectively. Also in this case, the up-rows contents are saved into the input buffer to avoid loss of data (1 nInstruction). $|x_s - x_i|$ terms are computed, saving the results in the output buffers and then moving them into the down-rows (3 nInstructions). Next, $|y_s - y_i|$ are performed (2 nInstruction), and $|x_s - x_i|$ and $|y_s - y_i|$ are summed together (1 nInstruction). Finally, the results are saved in the down-rows (1 nInstruction). The algorithm requires 8 steps to be completed.

3.5 Bitmap indexing

The bitmap indexing (BMP) is a technique for database fast search [26]. It consists of changing the data representation to perform the information search inside the database through simple bit-wise operations. Data are organized in bit arrays (called bitmaps), which can be true or false and each bit represents a field. If the record is associated with a field, the correspondent bit is set to true. The specific research example, implemented on the Hybrid-SIMD, takes in consideration a list of 8 students who could possibly take 2 exam versions with a grade from A to F. In particular, the algorithm answers the question: *how many students, who took the exam version 1, achieved mark A or B?* Each Hybrid-SIMD arithmetic memory row is initialized with a different data value. The smart row structure involves one more row interface called OneCounter, which implements the students count. At the beginning of the algorithm, the smart row containing the Mark A value takes as second operand the Mark B field and performs the bit-wise-OR operation. The result is stored at first in the output buffer and then in the down-row (2 nInstruction required). In this way, the down-row contains the bit array of students who achieved mark A or B. This data is then retaken as a second operand for a bit-wise-AND operation with the smart row containing the Exam version 1 data. Through the OneCounter, the number of ones in the result are counted and stored in the output buffer (2 nInstructions required). At last, the output buffer value is copied into the down-row (1 nInstruction required), where it is possible to find the final number of students compliant with the request. The algorithm requires 5 steps to be completed.

3.6 Mean & variance

This algorithm, called VAR, evaluates mean and variance for a set of N points {X}. These points are saved in the arithmetic memory rows and Hybrid-SIMD implements the following code:

```
sum1, sum2, sum3 = 0;
for i in 0 to N: sum1 += X(i);
mean = sum1/N;
for i in 0 to N:
    sum3 += (X(i) - mean);
    sum2 += (X(i) - mean)*(X(i) - mean);
end
variance = (sum2-sum3*sum3/N)/N;
```

There are 2 FOR loops involving the accumulation of the same term, requiring to exploit the up-row/down-row mechanism and to use the Hybrid-SIMD in sequential computing mode. The architecture cannot efficiently implement this algorithm, since it performs parallel computations, but it is a good metric to demonstrate benefits and limits of Hybrid-SIMD. In this case, the smart row contains an arithmetic memory row, a multiplier, an arithmetical right shifter, and two TemporaryStorages. The first FOR cycle requires 2N-1 nInstructions to be accomplished (sum computation and store in down-rows, for each smart row), propagating the sum1 in all the smart rows. The mean term is then computed with the right shifter and saved in the down-row with 2 nInstructions. For sum3, the same principle of sum1 is applied: the architecture computes X(i)-mean in parallel (1 nInstruction), saves the results in the input buffers (1 nInstruction), and exploits the up-row/down-row mechanism to propagate the sum (2N-1 nInstructions). In the last iteration, sum2 is saved in a TemporaryStorage to avoid data loss (1 nInstruction). This step requires 2N+2

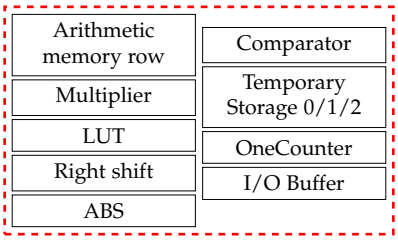

Fig. 9. Smart row structure implementing all the proposed benchmarks.

nInstructions. Until now, X(i)-mean terms are stored in the input buffers, so sum2 can be easily computed performing a power-2 operation exploiting Bi_Bi operand from Table 2 (1 nInstruction) and, similarly to sum1 and sum3, the sum is propagated until the last iteration (2N-1 nInstructions). Finally, variance is computed with 4 nInstructions. In total, the algorithm requires $6N + 7$ steps, resulting in being very inefficient.

4 Synthesis and Place&Route results

This section presents the results of Hybrid-SIMD with the algorithms described before. An advantage of Hybrid-SIMD architecture relies on the simplicity of the definition of design characteristics. The parameters that the user has to define are the number of rows (NRow), the number of smart rows (NSmartRow), the data parallelism (NBit) and the total number of blocks (nBlocks). All of them can be determined from the algorithm. From it, also the smart row structure is derived, as described in section 3. All the benchmarks require the arithmetic memory row and the I/O Buffer. In addition, the smart row contains one multiplier (required by DFT, K-NN, K-Means, VAR and MVM algorithms), one LUT (DFT), one OneCounter (BMP), three TemporaryStorages (K-Means, VAR), one right shifter (VAR), one comparator (K-Means) and one ABS row interface to compute the absolute value (K-NN). The smart row structure is depicted in Figure 9. The SIMD Array sizes are derived from the algorithm as well. For example, if MVM involves 512 elements, the total number of smart rows should be 512 to perform all the parallel computations. In all the tests, we fixed NRow = 1024, NSmartRow = 256, NBit = 32, nBlocks = 4. By multiplying NRow by Nbit, the memory size is obtained, and it is equal to 4kB.

4.1 Benchmark performance results

Power-area-timing values are extracted for each case study. In the estimations, we use Synopsys Design Compiler for synthesis together with Cadence Innovus for Place&Route procedure, employing 45nm CMOS technology. Synthesis powers are obtained with the worst-case scenario of maximum switching activity, while post Place&Route estimations are performed with back-annotation simulations with Mentor QuestaSim to improve accuracy.

Synthesis results are shown in Table 4: since the synthesized architecture is the same for all the proposed benchmarks, power, area and critical path values are equal in all the cases. For this reason, it could be useful to compare the results in terms of total energy, since it considers both the speed and power consumption of the Hybrid-SIMD. Energy is obtained as the product between the algorithm execution time (given by Critical path × Execution cycles) and the total power of the circuit. The VAR algorithm achieves the maximum energy value for the reasons already explained in subsection 3.6: VAR algorithm results in being very inefficient since the two FOR loops cannot be parallelized. On the other hand, the minimum energy is obtained in the BMP case since the algorithm is simpler than the other cases. In general, a low number of execution cycles brings many advantages since the algorithm is more efficiently implemented by our Hybrid-SIMD architecture, achieving a high degree of parallelization. After the synthesis, Place&Route and post Place&Route power estimation are performed to obtain even more precise values. Results are reported in Figure 10 (B) and Table 4 with a chosen clock period of 12 nanoseconds. The reasons behind this choice are explained in subsection 4.2. As expected, apart from DFT, powers result are better than synthesis one, since the activity factor of the circuit is lower than the worst-case scenario and the chosen clock frequency is smaller. At the same time, the critical path increases since a more realistic interconnections model is considered.

TABLE 4
Synthesis and Place&Route results using CMOS 45nm technology.

Test	Synthesis: worst-case switching					Place&Route backannotation	
	Power [mW]	Area [mm^2]	Critical path [ns]	Execution cycles [#clock cycles]	Total Energy [nJ]	Power [mW] @ 12ns	Critical path [ns]
DFT	198.09	2.15	4.89	1033	1000.63	212.90	6.79
K-Means				554	536.64	93.85	
K-NN				522	505.64	84.79	
MVM				546	528.89	114.20	
BMP				13	12.59	86.12	
VAR				1799	1742.62	181.40	

Synthesis power is obtained in worst-case switching activity scenario with a clock period equal to the critical path. Place&Route estimations are obtained with backannotation process with a clock period of 12ns.

4.2 Parametric sweeps

To determine the performance trend and a good trade-off between power and speed, parametric sweeps are performed. Power is evaluated with respect to SIMD Array size and different clock periods. A basic version of Hybrid-SIMD architecture is chosen, containing an arithmetic memory row, a multiplier, and the I/O Buffer in each smart row. The total number of smart rows varies with the NRow, and in particular, it is always equal to NRow/4. Results are shown in Figure 10 (A): as it is possible to see, power increases linearly with NRow, while decreases exponentially with smaller frequencies. The linear relation with NRow is a very good trend, indicating that a higher number of processing units wouldn't significantly degrade the performance, enabling more computations in parallel. The choice of a smaller clock period implies a larger power consumption. However, if the computations are executed in parallel, the execution time wouldn't be incisively degraded even if a lower frequency is chosen. From these considerations, a clock period of 12ns is chosen, that represent a good trade-off between power and speed.

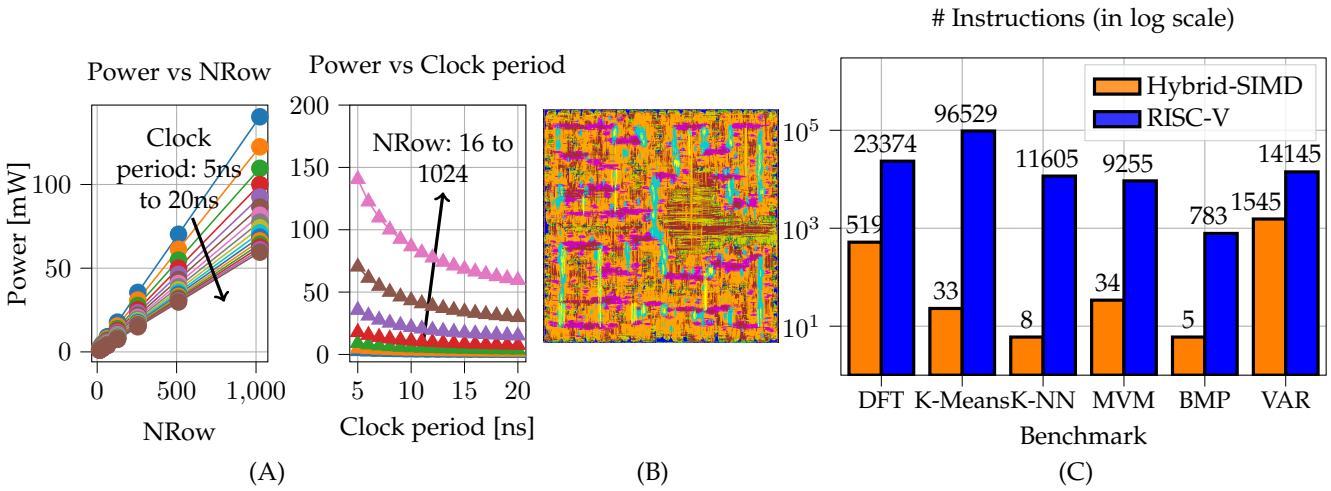

Fig. 10. (A) Power estimation with parametric sweeps of NRow and Clock period. The clock period is varied from 5ns to 20ns with a step of 1 ns. NRow starts from 16 and increases until 1024 with step of $\times 2$.(B) Place&Route snapshot of Hybrid-SIMD architecture for the considered benchmarks. (C) Number of instructions required for RISC-V/Hybrid-SIMD architectures.

5 Evaluation: Hybrid-SIMD impact on a von Neumann architecture

Until now, we discussed the benefits of Hybrid-SIMD architecture, which results to be modular and reconfigurable to fit different needs. One may ask: what is the impact of integrating the BvNC paradigm in a classical von Neumann system? What are the benefits of using Hybrid-SIMD as a coprocessor? To answer these questions, the same algorithms explained before are considered and implemented on a RISC-V processor, modeled with Gem5 simulator [27]. The RISC-V von Neumann architecture is made of two levels of caches, a DRAM and a single-core In-Order CPU. Level-1 is split into data and instruction caches of 4kB each. Level-2 cache is shared between data and instructions and has a size equal to 256kB. Finally, DRAM is a DDR3 @ 1600MHz of 512 MB. By studying this simple configuration, CPU-Memory system performance is evaluated and compared with Hybrid-SIMD architecture. Since memory is slower than the CPU, much time and energy are wasted only to wait for the data: Beyond von Neumann architectures try to minimize these quantities, which usually degrades classical architecture performance.

5.1 Number of required instructions

The first proposed comparison is the number of instructions required by the algorithms in both RISC-V and Hybrid-SIMD architectures. RISC-V implements the applications proposed in section 3 in C language. Source files are compiled and statically linked with `--static` option and finally, executed with a trace-based simulation to print a disassembled version of the algorithm (i.e., the instructions executed by the processor in @main section). An example of the output file provided by Gem5 is shown in Figure 11.

The number of instructions for the RISC-V is evaluated for each benchmark and compared with the total number of nInstructions of Hybrid-SIMD, that are estimated in Figure 10 (C). Tests consider N=256, Nc=3, and M=16, as deeply explained in subsection 5.2.

```
...
124: system.cpu: @_start+50 : c_lwsp a0, 0(sp)
125: system.cpu: @_start+52 : c_addi4spn a1, sp, 8
126: system.cpu: @_start+54 : c_li a2, 0
...
```

Fig. 11. Part of trace-based simulation of VAR algorithm: output example

5.2 Number of memory accesses

As expected, the total number of instructions required results to be far lower than the CPU: this happens because our architecture can perform SIMD computations, resulting in way faster than a CPU-based implementation. This consideration suggests that comparing the number of instructions alone is not sufficient to exploit Hybrid-SIMD versatility in a classical context. As a consequence, the number of memory accesses is estimated in both architectures. For RISC-V, Gem5 is used to provide statistics regarding the algorithm execution. These values can be found in the `stats.txt` output file. It contains the number of all the memory accesses from the beginning of the program until the end. In Hybrid-SIMD instead, memory accesses calculations consider that each smart row can hold up to 2 data (up-row/arithmetic memory row), so $max(\#\text{input data}) = \text{Nsmartrow} \times 2 = 512$. K-Means uses 256 coordinates, with Nc=3 centroids and Nc masks, required to identify the clusterIDs. Consequently, the total number of memory accesses for K-Means is 521 ($256 \times 2 + 3 \times 2 + 3$). In DFT there are 128 samples X_i to be memorized. These are written twice, one for the real and the other one for the imaginary part. For each smart row, X_i and i are stored in the up-row and arithmetic row, respectively, requiring 512 stores. The total number of memory accesses is also added by 2 to store (π,k) parameters, becoming 514. In K-NN, 256 points (X_i, Y_i) and 2 shared variables (X_c,Y_c) are used, reaching 514 stores. MVM is performed with $M \times N = 16 \times 16$ matrix and $N = 16$ column vector. The multiplication with the vector values is performed for each matrix row. The column vector has been replicated 16 times, requiring 512 memory writes in total. BMP requires to store each field. In the example

TABLE 5
RISC-V/Hybrid-SIMD memory accesses and energy comparisons

Test	Memory accesses (read+write)							Total energy [nJ]			
	RISC-V				Hybrid-SIMD			RISC-V		Hybrid-SIMD	Improvement [%]
	L1ICache	L1DCache	L2SCache	Total	uROM	SIMD Array	Total	Caches: accesses	Caches: leakage	Entire architecture	
DFT	30313	7132	56	37501	519	514	1033	2826.42	3877.57	2704.83	59.65
K-Means	118579	27091	388	146058	33	521	554	11077.19	15886.96	628.09	97.67
K-NN	15640	3208	52	18900	8	514	522	1434.12	2064.06	532.14	84.79
MVM	12729	3322	22	16073	34	512	546	1210.60	1652.69	752.54	73.72
BMP	986	318	16	1320	5	8	13	105.18	201.28	14.07	95.41
VAR	20564	6031	19	26614	1543	256	1799	1997.46	2656.06	4111.46	11.65

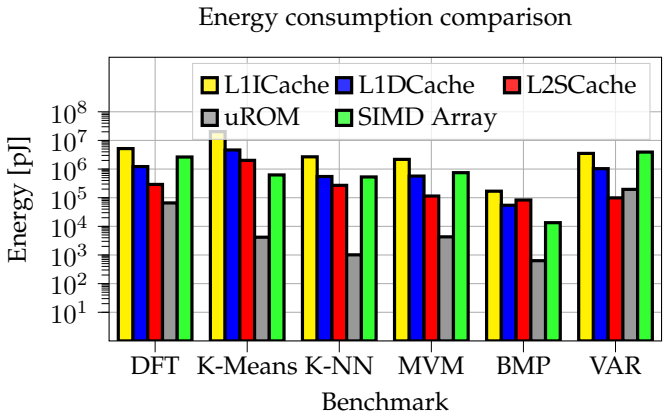

Fig. 12. Energy consumption comparison. RISC-V considers only memory access energy, while Hybrid-SIMD the entire architecture (memory accesses and computation)

presented in subsection 3.5, fields are associated with the exam type and the mark grade, so #Fiels is 8 of Nbit each, where Nbit, in this case, indicates the maximum number of samples X_i available. Finally, VAR computation considers X_i stored inside each arithmetic memory row. Hybrid-SIMD has 256 available smart rows, so the total number of memory accesses is equal to 256. For RISC-V, the results are shown in Table 5, where L1DCache, L1ICache, L2SCache indicate Level-1 Data/Instruction caches and Level-2 Shared cache, respectively. First level caches have a size of 4kB to be equal to SIMD Array dimensions. L2SCache has a size of 256kB. One of the most important contributions in the von Neumann bottleneck is the number of memory accesses. In all the cases, Hybrid-SIMD results to be more efficient than RISC-V, reducing memory accesses and appearing as a promising coprocessor in a classical system from this point of view.

5.3 Classical memory hierarchy consumption vs Hybrid-SIMD

Referring to Table 5, the energy required for each memory access can be estimated. Cacti [28] is exploited for this purpose. Cacti is a tool developed by HP Laboratories, that is able to model the performance of caches and memories precisely. We modeled L1D/ICache, L2SCache and uROM with Uniform Cache Access (UCA) and one read/one write ports with CMOS 45nm technology obtaining the results shown in Table 6, where "A" field indicates the memory associativity. Considering the energy/access and leakage power values, the caches energy consumption are estimated and compared with those obtained in Hybrid-SIMD for the studied algorithms. Leakage energy is computed as the total number of memory accesses of the cache multiplied by the leakage power and the clock period, that is fixed to 12 nanoseconds, as Hybrid-SIMD. On the other hand, Hybrid-SIMD energy is computed as the total power consumption multiplied by the execution time required to perform the memory operation (#SIMD Array accesses $\times\, t_{ck}$) plus the entire considered algorithm (#Required nInstructions $\times\, t_{ck}$). Power and clock values are extracted from the post Place&Route estimations reported in Table 4. Data placement overhead has a predominant role in the energy consumption calculation that has to be considered. The term "SIMD Array accesses" in the energy equation of Hybrid-SIMD considers the data placement overhead. It is expressed as the total number of memory accesses to write the required data in the memory. For each benchmark, this value is estimated in subsection 5.2 and reported in the "SIMD Array" column of Table 5. Previous assumptions imply a worst-case scenario for Hybrid-SIMD, since the entire architecture power consumption is evaluated (memory operations and computation). In contrast, the RISC-V estimations are performed in the best case, since only the caches energies are considered, neglecting the core consumption. The energy results for each memory are shown in Figure 12, where a visual comparison between RISC-V/Hybrid-SIMD is presented. Total energies are reported in Table 5. In terms of area, considering the values reported in Table 6 and Table 4, the Hybrid-SIMD occupies $\sim 10.29\times$ more area than the L1 caches of the same capacity. In the RISC-V architecture, the highest energy contribution is given by the instruction cache as shown in Figure 12, since it is accessed a lot in all the benchmarks. At first sight, the Hybrid-SIMD architecture contribution (given by the sum of uROM and SIMD Array energy consumptions) appears to be more energy efficient in the proposed benchmarks. VAR achieves the minimum value of 11.65% energy improvement because, as already explained in subsection 3.6, the Hybrid-SIMD architecture is very inefficient to implement sequential algorithms, implying a degradation of the energy and execution time. As

TABLE 6
L1-L2-uROM memories energy/access and access time

Memory	Size	A	Energy/Access [nJ]			Leakage power [mW]	Access time [ns]	Area [mm^2]
			Read	Write	Avg			
L1I/D Cache	4kB	2	0.073	0.076	0.075	8.042	0.541	0.209
L2SCache	256kB	8	0.454	0.507	0.480	392.939	2.125	3.009
uROM (SRAM)	4kB	1	0.037	0.046	0.041	7.101	0.306	0.119

expected, the second worst value is achieved in DFT case because it is slower than the other ones, but Hybrid-SIMD is still energetically convenient. Other benchmarks highlight very promising performance results as shown in Figure 12 and confirmed in Table 5, with an energy improvement of at least 73% in comparison to RISC-V memory hierarchy. Moreover, Hybrid-SIMD memory accesses are drastically reduced compared to the caches since the SIMD Array is capable of both storing and computing data at the same time, decreasing data traffic. In conclusion, we can say that employing Hybrid-SIMD as a coprocessor architecture for parallel algorithms in a von Neumann classical system can bring a lot of advantages, both in terms of memory accesses and energy efficiency. Using it as a coprocessor enables the CPU to delegate massively parallel jobs to Hybrid-SIMD, improving performance.

6 Conclusion

In this paper, we presented Hybrid-SIMD, a modular BvNC coprocessor architecture capable of reducing memory accesses and improving energy efficiency in a classical von Neumann architecture for the considered benchmarks. A strong point of Hybrid-SIMD is the modularity, since the smart row structure can be changed according to the algorithm. Thus, the user can insert the row interfaces required by an unique algorithm (application-specific approach) or by a given set of benchmarks (generic approach). With the generic approach, Hybrid-SIMD obtains an energy improvement up to 97.67% with respect to the studied memory hierarchy of RISC-V processor. Massively parallel algorithms are efficiently implemented on Hybrid-SIMD, gaining a considerable speed-up and a lower number of memory accesses. On the other hand, sequential algorithms (like VAR) are not efficiently executed on Hybrid-SIMD, implying that standard CPU-based architectures are more suitable to perform this kind of computations. In conclusion, using Hybrid-SIMD as a coprocessor bring remarkable improvements in a classical von Neumann architecture, especially in case of parallel computations, reducing the von Neumann bottleneck.

References

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2017.

[2] K. C. Akyel, H.-P. Charles, J. Mottin, B. Giraud, G. Suraci, S. Thuries, and J.-P. Noel, "Drc 2: Dynamically reconfigurable computing circuit based on memory architecture," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*. IEEE, 2016, pp. 1–8.

[3] Z. Lin, H. Zhan, X. Li, C. Peng, W. Lu, X. Wu, and J. Chen, "In-memory computing with double word lines and three read ports for four operands," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 5, pp. 1316–1320, 2020.

[4] H. Jia, H. Valavi, Y. Tang, J. Zhang, and N. Verma, "A programmable heterogeneous microprocessor based on bit-scalable in-memory computing," *IEEE Journal of Solid-State Circuits*, vol. 55, no. 9, pp. 2609–2621, 2020.

[5] M. F. Ali, A. Jaiswal, and K. Roy, "In-memory low-cost bit-serial addition using commodity dram technology," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 1, pp. 155–165, 2019.

[6] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 273–287.

[7] A. Jaiswal, I. Chakraborty, A. Agrawal, and K. Roy, "8t sram cell as a multibit dot-product engine for beyond von neumann computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2556–2567, 2019.

[8] W. A. Simon, Y. M. Qureshi, M. Rios, A. Levisse, M. Zapater, and D. Atienza, "An in-cache computing architecture for edge devices," *IEEE Transactions on Computers*, 2020.

[9] L. Yavits, A. Morad, and R. Ginosar, "Computer architecture with associative processor replacing last-level cache and simd accelerator," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 368–381, 2013.

[10] W. Choi, K. Jeong, K. Choi, K. Lee, and J. Park, "Content addressable memory based binarized neural network accelerator using time-domain signal processing," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[11] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*. Springer, 2016, pp. 525–542.

[12] M. Durlam, P. Naji, M. DeHerrera, S. Tehrani, G. Kerszykowski, and K. Kyler, "Nonvolatile ram based on magnetic tunnel junction elements," in *2000 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No. 00CH37056)*. IEEE, 2000, pp. 130–131.

[13] A. S. Rakin, S. Angizi, Z. He, and D. Fan, "Pim-tgan: A processing-in-memory accelerator for ternary generative adversarial networks," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 266–273.

[14] A. Roohi, S. Angizi, D. Fan, and R. F. DeMara, "Processing-in-memory acceleration of convolutional neural networks for energy-effciency, and power-intermittency resilience," in *20th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2019, pp. 8–13.

[15] H. Wang and X. Yan, "Overview of resistive random access memory (rram): Materials, filament mechanisms, performance optimization, and prospects," *physica status solidi (RRL)–Rapid Research Letters*, vol. 13, no. 9, p. 1900073, 2019.

[16] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[17] O. Krestinskaya and A. P. James, "Binary weighted memristive analog deep neural network for near-sensor edge processing," in *2018 IEEE 18th International Conference on Nanotechnology (IEEE-NANO)*. IEEE, 2018, pp. 1–4.

[18] X. Wang, M. A. Zidan, and W. D. Lu, "A crossbar-based in-memory computing architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 4224–4232, 2020.

[19] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature Nanotechnology*, pp. 1–16, 2020.

[20] I. Giannopoulos, A. Sebastian, M. Le Gallo, V. Jonnalagadda, M. Sousa, M. Boon, and E. Eleftheriou, "8-bit precision in-memory multiplication with projected phase-change memory," in *2018 IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2018, pp. 27–7.

[21] G. Karunaratne, M. Le Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian, "In-memory hyperdimensional computing," *Nature Electronics*, vol. 3, no. 6, pp. 327–337, 2020.

[22] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3d-stacked dram," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 131–143, 2015.

[23] A. Morad, L. Yavits, and R. Ginosar, "Gp-simd processing-in-

memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, pp. 1–26, 2015.
[24] S. Basalama, A. Panahi, A.-T. Ishimwe, and D. Andrews, "Spar-2: A simd processor array for machine learning in iot devices," in *2020 3rd International Conference on Data Intelligence and Security (ICDIS)*, 2020, pp. 141–147.
[25] X. Jin and J. Han, *K-Means Clustering*. Boston, MA: Springer US, 2010, pp. 563–564. [Online]. Available: https://doi.org/10.1007/978-0-387-30164-8_425
[26] P. Lane, V. Schupmann, and I. Stuart, "Oracle database data warehousing guide," *10g Release*, vol. 2, no. 10.2, 2005.
[27] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.
[28] M. et al., "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

Andrea Coluccio He received his Master Degree in Electronic Engineering, specialization in Electronic Systems, from the Polytechnic University of Turin (Polito) in 2019. He is currently a Ph.D. student in Electronic Engineering at Polito. His research interests are Logic-in-Memory and Beyond von Neumann computing paradigms implemented with standard and emerging technologies.

Umberto Casale Umberto Casale was born in Avellino, Italy, in 1996. He received the B.Sc. degree in Electronic Engineer at the Politecnico di Torino (07/2015-09/2018). He received the M.Sc. degree in Electronic Engineer at the Politecnico di Torino (09/2018-07/2020). He received M.Sc. in Electric and Computer Engineering at the University Of Illinois at Chicago for a Double Degrees Project (09/2019-07/2020). He is currently working as Digital Design Verification Engineer at Apple, which he joined in September 2020.

Angela Guastamacchia She attained the M.Sc. degree in Electronic Engineering, with specialization in Embedded Systems, at the Politecnico di Torino in 2021. She is currently enrolled in the Bioengineering and Medical-Surgical Sciences Ph.D. course jointly offered by Politecnico di Torino and Università degli studi di Torino. Her present research focuses on the validation of a laboratory for the reproduction of ecological audiovisual virtual scenes aimed at biomedical applications in the hearing aided field.

Giovanna Turvani She received the M.Sc. degree with honours (Magna Cum Laude) in Electronic Engineering in 2012 and the Ph.D. degree from the Politecnico di Torino. She is currently Assistant Professor at Politecnico di Torino. Her interests include CAD Tools development for non-CMOS nanocomputing, architectural design for field-coupled nanocomputing and high-level device modelling for Quantum Computing and hardware systems for microwave imaging-based techniques for biomedical applications and for food quality monitoring.

Marco Vacca Marco Vacca is Associate Professor at the Department of Electronics and Telecommunications of Politecnico di Torino. His research interests include innovative and unconventional computer architectures and beyond-CMOS technologies, particularly magnetic devices like the racetrack memories. Another research branch is focused in the development of intelligent systems for industry and agricolture 4.0.

Massimo Ruo Roch He achieved the Electronics Engineering degree in 1990 and the Ph.D. degree in 1993 from the Politecnico di Torino. Since 2002, he has been a full-time Researcher at the Politecnico di Torino. His research interests include digital design of application-specific computing architectures, high-speed telecommunications, and digital television. Recent activities include design and modeling of MPSoCs, embedded systems for bio applications, and cloud-based systems for e-learning.

Maurizio Zamboni He received the Electronics Eng. and the Ph.D. degrees in 1983 and in 1988 from the Politecnico di Torino, respectively, where he is currently a Full Professor. His research activity focuses on multiprocessor architectures design, in IC optimization for artificial intelligence, telecommunication, low-power circuits, and innovative beyond CMOS technologies.

Mariagrazia Graziano She received the Dr.Eng. degree and the Ph.D in Electronics Engineering from the Politecnico di Torino, Italy, in 1997 and 2001, respectively. Since 2002 she is Assistant Professor at the Politecnico di Torino. Since 2008 she is adjunct Faculty at the University of Illinois at Chicago and since 2014 she is a Marie-Curie fellow at the London Centre for Nanoelectronics. She works on "beyond CMOS" devices, circuits and architectures.